

**System and Method for Managing  
a Plurality of Processors as Devices**

**EL977166451US****BACKGROUND OF THE INVENTION****1. Technical Field**

5           The present invention relates in general to a system and method for managing a plurality of processors as devices. In particular, the present invention relates to a system and a method for applications to directly access a plurality of processors using device-oriented instructions.

10           **2. Description of the Related Art**

          In most multiprocessor systems, access to the processors is facilitated through the operating system, which is executing on one or more of the processors (typically the main processor). Applications typically are  
15       unaware of a computer system's processor configuration. When an application requests the execution of a process, the operating system determines which of the multiple processors is to execute the process. In addition, the operating system determines when and if to swap a process  
20       executing on a particular processor for another. A programmer typically does not have direct control of a computer system's processor resources.

          There are many instances when a programmer would like to have direct access to each of the available processors.  
25       In gaming applications, for example, programmers are always searching for the most direct control over a computer system's resources and in particular a computer system's multiple processors.

What is needed, therefore, is a system and method that could provide a programmer with such direct access to the multiple processors of a computer system. The system and method should provide a programmer with access to the  
5 multiple processors similar to the access a programmer has to other devices connected to the computer system such as printers, disk drives, and the like.

**SUMMARY**

5 It has been discovered that the aforementioned challenges can be addressed by a method and a system for managing a computer system's multiple processors as devices. Applications, through the operating system, access the multiple processors of a computer system using processor device modules similar to the way applications access other devices connected to the computer system such as disk drives, printers, etc.

10 An operating system is executing on at least one first processor of the computer system. A device module corresponding to one or more secondary processors is loaded into the operating system. The device module facilitates a communication between an application seeking access to one  
15 or more of the secondary processors and the secondary processors. The device module extends the operating system such that the operating system can execute the device-oriented instructions. The operating system is configured to create an instructions block for the secondary  
20 processors to use in processing the application's task.

Device-like access is set up for one or more of the secondary processors. The device-like access may be used by an application to access one or more of the secondary processors. Similar access, for example, may be used by an  
25 application to access other devices connected to the computer system, such as disk drives, printers, etc. Information about the set up device-like access for the secondary processors is stored in a device configuration file.

An application seeking access to the one or more secondary processors is loaded by the operating system. The application code includes device-oriented instructions adapted to control one or more of the secondary processors.

5 The loaded device module determines for which secondary processor the application's instructions are intended. The determination may be performed using information from the device configuration file. The application may also provide the one or more secondary processors with data to  
10 be processed. The one or more secondary processors process the data according to the instructions provided by the application and notify the operating system of the end of the processing.

The foregoing is a summary and thus contains, by  
15 necessity, simplifications, generalizations, and omissions of detail; consequently, those skilled in the art will appreciate that the summary is illustrative only and is not intended to be in any way limiting. Other aspects, inventive features, and advantages of the present  
20 invention, as defined solely by the claims, will become apparent in the non-limiting detailed description set forth below.

**BRIEF DESCRIPTION OF THE DRAWINGS**

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings. The use of the same reference symbols in different drawings indicates similar or identical items.

**Figure 1** illustrates -the overall architecture of a computer network in accordance with the present invention;

**Figure 2** is a diagram illustrating the structure of a processing unit (PU) in accordance with the present invention;

**Figure 3** is a diagram illustrating the structure of a broadband engine (BE) in accordance with the present invention;

**Figure 4** is a diagram illustrating the structure of an synergistic processing unit (SPU) in accordance with the present invention;

**Figure 5** is a diagram illustrating the structure of a processing unit, visualizer (VS) and an optical interface in accordance with the present invention;

**Figure 6** is a diagram illustrating one combination of processing units in accordance with the present invention;

**Figure 7** illustrates another combination of processing units in accordance with the present invention;

**Figure 8** illustrates yet another combination of processing units in accordance with the present invention;

**Figure 9** illustrates yet another combination of processing units in accordance with the present invention;

5       **Figure 10** illustrates yet another combination of processing units in accordance with the present invention;

**Figure 11A** illustrates the integration of optical interfaces within a chip package in accordance with the present invention;

10       **Figure 11B** is a diagram of one configuration of processors using the optical interfaces of **Figure 11A**;

**Figure 11C** is a diagram of another configuration of processors using the optical interfaces of **Figure 11A**;

15       **Figure 12A** illustrates the structure of a memory system in accordance with the present invention;

**Figure 12B** illustrates the writing of data from a first broadband engine to a second broadband engine in accordance with the present invention;

20       **Figure 13** is a diagram of the structure of a shared memory for a processing unit in accordance with the present invention;

**Figure 14A** illustrates one structure for a bank of the memory shown in **Figure 13**;

**Figure 14B** illustrates another structure for a bank of the memory shown in **Figure 13**;

**Figure 15** illustrates a structure for a direct memory access controller in accordance with the present invention;

5        **Figure 16** illustrates an alternative structure for a direct memory access controller in accordance with the present invention;

**Figures; 17-31** illustrate the operation of data synchronization in accordance with the present invention;

10        **Figure 32** is a three-state memory diagram illustrating the various states of a memory location in accordance with the data synchronization scheme of the-present invention;

15        **Figure 33** illustrates the structure of a key control table for a hardware sandbox in accordance with the present invention;

**Figure 34** illustrates a scheme for storing memory access keys for a hardware sandbox in accordance with the present invention;

20        **Figure 35** illustrates the structure of a memory access control table for a hardware sandbox in accordance with the present invention;

**Figure 36** is a flow diagram of the steps for accessing a memory sandbox using the key control table of **Figure 33** and the memory access control table of **Figure 35**;

**Figure 37** illustrates the structure of a software cell in accordance with the present invention;

**Figure 38** is a flow diagram of the steps for issuing remote procedure calls to SPUs in accordance with the present invention;

**Figure 39** illustrates the structure of a dedicated pipeline for processing streaming data in accordance with the present invention;

**Figure 40** is a flow diagram of the steps performed by the dedicated pipeline of **Figure 39** in the processing of streaming data in accordance with the present invention;

**Figure 41** illustrates an alternative structure for a dedicated pipeline for the processing of streaming data in accordance with the present invention;

**Figure 42** illustrates a scheme for an absolute timer for coordinating the parallel processing of applications and data by SPUs in accordance with the present invention;

**Figure 43** is a block diagram illustrating a processing element having a main processor and a plurality of secondary processors sharing a system memory;

**Figure 44** is a block diagram illustrating a processor element having a main processor and a plurality of secondary processors sharing a common system memory;



**Figure 45A** is a table illustrating a device configuration file that contains device access paths and corresponding devices;

5       **Figure 45B** is a table illustrating example application code for accessing a system's multiple processors;

**Figure 46** is a flowchart illustrating a method for setting up device-like access to a system's multiple processors;

10       **Figure 47** is a flowchart illustrating a method by which an application requests device-like access to a processor in a multiprocessor system;

15       **Figure 48** is a flowchart illustrating a method by which an operating system kernel receives a request from an application to access a processor in a multi-processor system and facilitates access to the processor; and

**Figure 49** is a flowchart illustrating a method with which a processor processes an application's request to access a processor.

**DETAILED DESCRIPTION**

The following is intended to provide a detailed description of an example of the invention and should not be taken to be limiting of the invention itself. Rather,  
5 any number of variations may fall within the scope of the invention defined in the claims following the description.

The overall architecture for a computer system **101** in accordance with the present invention is shown in **Figure 1**.

As illustrated in this figure, system **101** includes  
10 network **104** to which is connected a plurality of computers and computing devices. Network **104** can be a LAN, a global network, such as the Internet, or any other computer network.

The computers and computing devices connected to  
15 network **104** (the network's "members") include, e.g., client computers **106**, server computers **108**, personal digital assistants (PDAs) **110**, digital television (DTV) **112** and other wired or wireless computers and computing devices. The processors employed by the members of network **104** are  
20 constructed from the same common computing module. These processors also preferably all have the same ISA and perform processing in accordance with the same instruction set. The number of modules included within any particular processor depends upon the processing power required by  
25 that processor.

For example, since servers **108** of system **101** perform more processing of data and applications than clients **106**,

5 servers **108** contain more computing modules than clients **106**. PDAs **110**, on the other hand, perform the least amount of processing. PDAs **110**, therefore, contain the smallest number of computing modules. DTV **112** performs a level of processing between that of clients **106** and servers **108**. DTV **112**, therefore, contains a number of computing modules between that of clients **106** and servers **108**. As discussed below, each computing module contains a processing controller and a plurality of identical processing units for performing parallel processing of the data and applications transmitted over network **104**.

15 This homogeneous configuration for system **101** facilitates adaptability, processing speed and processing efficiency. Because each member of system **101** performs processing using one or more (or some fraction) of the same computing module, the particular computer or computing device performing the actual processing of data and applications is unimportant. The processing of a particular application and data, moreover, can be shared among the network's members. By uniquely identifying the cells comprising the data and applications processed by system **101** throughout the system, the processing results can be transmitted to the computer or computing device requesting the processing regardless of where this processing occurred. Because the modules performing this processing have a common structure and employ a common ISA, the computational burdens of an added layer of software to achieve compatibility among the processors is avoided. This architecture and programming model facilitates the processing speed necessary to execute, e.g., real-time, multimedia applications.

To take further advantage of the processing speeds and efficiencies facilitated by system **101**, the data and applications processed by this system are packaged into uniquely identified, uniformly formatted software cells

5     **102**. Each software cell **102** contains, or can contain, both applications and data. Each software cell also contains an ID to globally identify the cell throughout network **104** and system **101**. This uniformity of structure for the software cells, and the software cells' unique identification

10    throughout the network, facilitates the processing of applications and data on any computer or computing device of the network. For example, a client **106** may formulate a software cell **102** but, because of the limited processing capabilities of client **106**, transmit this software cell to

15    a server **108** for processing. Software cells can migrate, therefore, throughout network **104** for processing on the basis of the availability of processing resources on the network.

The homogeneous structure of processors and software

20    cells of system **101** also avoids many of the problems of today's heterogeneous networks. For example, inefficient programming models which seek to permit processing of applications on any ISA using any instruction set, e.g., virtual machines such as the Java virtual machine, are

25    avoided. System **101**, therefore, can implement broadband processing far more effectively and efficiently than today's networks.

The basic processing module for all members of network **104** is the processing unit (PU). **Figure 2** illustrates the

30    structure of a PU. As shown in this figure, PE **201**

comprises a processing unit (PU) **203**, a direct memory access controller (DMAC) **205** and a plurality of synergistic processing units (SPUs), namely, SPU **207**, SPU **209**, SPU **211**, SPU **213**, SPU **215**, SPU **217**, SPU **219** and SPU **221**. A local PE bus **223** transmits data and applications among the SPUs, DMAC **205** and PU **203**. Local PE bus **223** can have, e.g., a conventional architecture or be implemented as a packet switch network. Implementation as a packet switch network, while requiring more hardware, increases available bandwidth.

PE **201** can be constructed using various methods for implementing digital logic. PE **201** preferably is constructed, however, as a single integrated circuit employing a complementary metal oxide semiconductor (CMOS) on a silicon substrate. Alternative materials for substrates include gallium arsenide, gallium aluminum arsenide and other so-called III-B compounds employing a wide variety of dopants. PE **201** also could be implemented using superconducting material, e.g., rapid single-flux-quantum (RSFQ) logic.

PE **201** is closely associated with a dynamic random access memory (DRAM) **225** through a high bandwidth memory connection **227**. DRAM **225** functions as the main memory for PE **201**. Although a DRAM **225** preferably is a dynamic random access memory, DRAM **225** could be implemented using other means, e.g., as a static random access memory (SRAM), a magnetic random access memory (MRAM), an optical memory or a holographic memory. DMAC **205** facilitates the transfer of data between DRAM **225** and the SPUs and PU of PE **201**. As further discussed below, DMAC **205** designates for each SPU

an exclusive area in DRAM **225** into which only the SPU can write data and from which only the SPU can read data. This exclusive area is designated a "sandbox."

5        PU **203** can be, e.g., a standard processor capable of stand-alone processing of data and applications. In operation, PU **203** schedules and orchestrates the processing of data and applications by the SPUs. The SPUs preferably are single instruction, multiple data (SIMD) processors. Under the control of PU **203**, the SPUs perform the  
10        processing of these data and applications in a parallel and independent manner. DMAC **205** controls accesses by PU **203** and the SPUs to the data and applications stored in the shared DRAM **225**. Although PE **201** preferably includes eight SPUs, a greater or lesser number of SPUs can be employed in  
15        a PU depending upon the processing power required. Also, a number of PUs, such as PE **201**, may be joined or packaged together to provide enhanced processing power.

For example, as shown in **Figure 3**, four PUs may be packaged or joined together, e.g., within one or more chip  
20        packages, to form a single processor for a member of network **104**. This configuration is designated a broadband engine (BE). As shown in **Figure 3**, BE **301** contains four PUs, namely, PE **303**, PE **305**, PE **307** and PE **309**. Communications among these PUs are over BE bus **311**. Broad  
25        bandwidth memory connection **313** provides communication between shared DRAM **315** and these PUs. In lieu of BE bus **311**, communications among the PUs of BE **301** can occur through DRAM **315** and this memory connection.

Input/output (I/O) interface **317** and external bus **319** provide communications between broadband engine **301** and the other members of network **104**. Each PU of BE **301** performs processing of data and applications in a parallel and independent manner analogous to the parallel and independent processing of applications and data performed by the SPUs of a PU.

**Figure 4** illustrates the structure of an SPU. SPU **402** includes local memory **406**, registers **410**, four floating point units **412** and four integer units **414**. Again, however, depending upon the processing power required, a greater or lesser number of floating points units **412** and integer units **414** can be employed. In a preferred embodiment, local memory **406** contains 128 kilobytes of storage, and the capacity of registers **410** is 128.times.128 bits. Floating point units **412** preferably operate at a speed of 32 billion floating point operations per second (32 GFLOPS), and integer units **414** preferably operate at a speed of 32 billion operations per second (32 GOPS).

Local memory **406** is not a cache memory. Local memory **406** is preferably constructed as an SRAM. Cache coherency support for an SPU is unnecessary. A PU may require cache coherency support for direct memory accesses initiated by the PU. Cache coherency support is not required, however, for direct memory accesses initiated by an SPU or for accesses from and to external devices.

SPU **402** further includes bus **404** for transmitting applications and data to and from the SPU. In a preferred embodiment, this bus is 1,024 bits wide. SPU **402** further

includes internal busses **408**, **420** and **418**. In a preferred embodiment, bus **408** has a width of 256 bits and provides communications between local memory **406** and registers **410**. Busses **420** and **418** provide communications between, respectively, registers **410** and floating point units **412**, and registers **410** and integer units **414**. In a preferred embodiment, the width of busses **418** and **420** from registers **410** to the floating point or integer units is 384 bits, and the width of busses **418** and **420** from the floating point or integer units to registers **410** is 128 bits. The larger width of these busses from registers **410** to the floating point or integer units than from these units to registers **410** accommodates the larger data flow from registers **410** during processing. A maximum of three words are needed for each calculation. The result of each calculation, however, normally is only one word.

**Figures. 5-10** further illustrate the modular structure of the processors of the members of network **104**. For example, as shown in **Figure 5**, a processor may comprise a single PU **502**. As discussed above, this PU typically comprises a PU, DMAC and eight SPUs. Each SPU includes local storage (LS) . On the other hand, a processor may comprise the structure of visualizer (VS) **505**. As shown in **Figure 5**, VS **505** comprises PU **512**, DMAC **514** and four SPUs, namely, SPU **516**, SPU **518**, SPU **520** and SPU **522**. The space within the chip package normally occupied by the other four SPUs of a PU is occupied in this case by pixel engine **508**, image cache **510** and cathode ray tube controller (CRTC) **504**. Depending upon the speed of communications required for PU **502** or VS **505**, optical interface **506** also may be included on the chip package.



Using this standardized, modular structure, numerous other variations of processors can be constructed easily and efficiently. For example, the processor shown in **Figure 6** comprises two chip packages, namely, chip package **602** comprising a BE and chip package **604** comprising four VSs. Input/output (I/O) **606** provides an interface between the BE of chip package **602** and network **104**. Bus **608** provides communications between chip package **602** and chip package **604**. Input output processor (IOP) **610** controls the flow of data into and out of I/O **606**. I/O **606** may be fabricated as an application specific integrated circuit (ASIC). The output from the VSs is video signal **612**.

**Figure 7** illustrates a chip package for a BE **702** with two optical interfaces **704** and **706** for providing ultra high speed communications to the other members of network **104** (or other chip packages locally connected). BE **702** can function as, e.g., a server on network **104**.

The chip package of **Figure 8** comprises two PEs **802** and **804** and two VSs **806** and **808**. An I/O **810** provides an interface between the chip package and network **104**. The output from the chip package is a video signal. This configuration may function as, e.g., a graphics work station.

**Figure 9** illustrates yet another configuration. This configuration contains one-half of the processing power of the configuration illustrated in **Figure 8**. Instead of two PUs, one PE **902** is provided, and instead of two VSs, one VS **904** is provided. I/O **906** has one-half the bandwidth of the

I/O illustrated in **Figure 8**. Such a processor also may function, however, as a graphics work station.

A final configuration is shown in **Figure 10**. This processor consists of only a single VS **1002** and an I/O  
5 **1004**. This configuration may function as, e.g., a PDA.

**Figure 11A** illustrates the integration of optical interfaces into a chip package of a processor of network  
10 **104**. These optical interfaces convert optical signals to electrical signals and electrical signals to optical signals and can be constructed from a variety of materials including, e.g., gallium arsinide, aluminum gallium arsinide, germanium and other elements or compounds. As shown in this figure, optical interfaces **1104** and **1106** are fabricated on the chip package of BE **1102**. BE bus **1108**  
15 provides communication among the PUs of BE **1102**, namely, PE **1110**, PE **1112**, PE **1114**, PE **1116**, and these optical interfaces. Optical interface **1104** includes two ports, namely, port **1118** and port **1120**, and optical interface **1106** also includes two ports, namely, port **1122** and port **1124**.  
20 Ports **1118**, **1120**, **1122** and **1124** are connected to, respectively, optical wave guides **1126**, **1128**, **1130** and **1132**. Optical signals are transmitted to and from BE **1102** through these optical wave guides via the ports of optical interfaces **1104** and **1106**.

25 plurality of BEs can be connected together in various configurations using such optical wave guides and the four optical ports of each BE. For example, as shown in **Figure 11B**, two or more BEs, e.g., BE **1152**, BE **1154** and BE **1156**, can be connected serially through such optical ports. In

this example, optical interface **1166** of BE **1152** is connected through its optical ports to the optical ports of optical interface **1160** of BE **1154**. In a similar manner, the optical ports of optical interface **1162** on BE **1154** are connected to the optical ports of optical interface **1164** of BE **1156**.

A matrix configuration is illustrated in **Figure 11C**. In this configuration, the optical interface of each BE is connected to two other BEs. As shown in this figure, one of the optical ports of optical interface **1188** of BE **1172** is connected to an optical port of optical interface **1182** of BE **1176**. The other optical port of optical interface **1188** is connected to an optical port of optical interface **1184** of BE **1178**. In a similar manner, one optical port of optical interface **1190** of BE **1174** is connected to the other optical port of optical interface **1184** of BE **1178**. The other optical port of optical interface **1190** is connected to an optical port of optical interface **1186** of BE **1180**. This matrix configuration can be extended in a similar manner to other BEs.

Using either a serial configuration or a matrix configuration, a processor for network **104** can be constructed of any desired size and power. Of course, additional ports can be added to the optical interfaces of the BEs, or to processors having a greater or lesser number of PUs than a BE, to form other configurations.

**Figure 12A** illustrates the control system and structure for the DRAM of a BE. A similar control system and structure is employed in processors having other sizes

and containing more or less PUs. As shown in this figure, a cross-bar switch connects each DMAC **1210** of the four PUs comprising BE **1201** to eight bank controls **1206**. Each bank control **1206** controls eight banks **1208** (only four are shown in the figure) of DRAM **1204**. DRAM **1204**, therefore, comprises a total of sixty-four banks. In a preferred embodiment, DRAM **1204** has a capacity of 64 megabytes, and each bank has a capacity of 1 megabyte. The smallest addressable unit within each bank, in this preferred embodiment, is a block of **1024** bits.

BE **1201** also includes switch unit **1212**. Switch unit **1212** enables other SPUs on BEs closely coupled to BE **1201** to access DRAM **1204**. A second BE, therefore, can be closely coupled to a first BE, and each SPU of each BE can address twice the number of memory locations normally accessible to an SPU. The direct reading or writing of data from or to the DRAM of a first BE from or to the DRAM of a second BE can occur through a switch unit such as switch unit **1212**.

For example, as shown in **Figure 12B**, to accomplish such writing, the SPU of a first BE, e.g., SPU **1220** of BE **1222**, issues a write command to a memory location of a DRAM of a second BE, e.g., DRAM **1228** of BE **1226** (rather than, as in the usual case, to DRAM **1224** of BE **1222**). DMAC **1230** of BE **1222** sends the write command through cross-bar switch **1221** to bank control **1234**, and bank control **1234** transmits the command to an external port **1232** connected to bank control **1234**. DMAC **1238** of BE **1226** receives the write command and transfers this command to switch unit **1240** of BE **1226**. Switch unit **1240** identifies the DRAM address contained in the write command and sends the data for

storage in this address through bank control **1242** of BE **1226** to bank **1244** of DRAM **1228**. Switch unit **1240**, therefore, enables both DRAM **1224** and DRAM **1228** to function as a single memory space for the SPUs of BE **1226**.

5           **Figure 13** shows the configuration of the sixty-four banks of a DRAM. These banks are arranged into eight rows, namely, rows **1302**, **1304**, **1306**, **1308**, **1310**, **1312**, **1314** and **1316** and eight columns, namely, columns **1320**, **1322**, **1324**, **1326**, **1328**, **1330**, **1332** and **1334**. Each row is controlled by  
10       a bank controller. Each bank controller, therefore, controls eight megabytes of memory.

**Figures. 14A** and **14B** illustrate different configurations for storing and accessing the smallest addressable memory unit of a DRAM, e.g., a block of **1024**  
15       bits. In **Figure 14A**, DMAC **1402** stores in a single bank **1404** eight **1024** bit blocks **1406**. In **Figure 14B**, on the other hand, while DMAC **1412** reads and writes blocks of data containing **1024** bits, these blocks are interleaved between two banks, namely, bank **1414** and bank **1416**. Each of these  
20       banks, therefore, contains sixteen blocks of data, and each block of data contains 512 bits. This interleaving can facilitate faster accessing of the DRAM and is useful in the processing of certain applications.

**Figure 15** illustrates the architecture for a DMAC **1504**  
25       within a PE. As illustrated in this figure, the structural hardware comprising DMAC **1506** is distributed throughout the PE such that each SPU **1502** has direct access to a structural node **1504** of DMAC **1506**. Each node executes the

logic appropriate for memory accesses by the SPU to which the node has direct access.

**Figure 16** shows an alternative embodiment of the DMAC, namely, a non-distributed architecture. In this case, the structural hardware of DMAC **1606** is centralized. SPUs **1602** and PU **1604** communicate with DMAC **1606** via local PE bus **1607**. DMAC **1606** is connected through a cross-bar switch to a bus **1608**. Bus **1608** is connected to DRAM **1610**.

As discussed above, all of the multiple SPUs of a PU can independently access data in the shared DRAM. As a result, a first SPU could be operating upon particular data in its local storage at a time during which a second SPU requests these data. If the data were provided to the second SPU at that time from the shared DRAM, the data could be invalid because of the first SPU's ongoing processing which could change the data's value. If the second processor received the data from the shared DRAM at that time, therefore, the second processor could generate an erroneous result. For example, the data could be a specific value for a global variable. If the first processor changed that value during its processing, the second processor would receive an outdated value. A scheme is necessary, therefore, to synchronize the SPUs' reading and writing of data from and to memory locations within the shared DRAM. This scheme must prevent the reading of data from a memory location upon which another SPU currently is operating in its local storage and, therefore, which are not current, and the writing of data into a memory location storing current data.

To overcome these problems, for each addressable memory location of the DRAM, an additional segment of memory is allocated in the DRAM for storing status information relating to the data stored in the memory location. This status information includes a full/empty (F/E) bit, the identification of an SPU (SPU ID) requesting data from the memory location and the address of the SPU's local storage (LS address) to which the requested data should be read. An addressable memory location of the DRAM can be of any size. In a preferred embodiment, this size is **1024** bits.

The setting of the F/E bit to 1 indicates that the data stored in the associated memory location are current. The setting of the F/E bit to 0, on the other hand, indicates that the data stored in the associated memory location are not current. If an SPU requests the data when this bit is set to 0, the SPU is prevented from immediately reading the data. In this case, an SPU ID identifying the SPU requesting the data, and an LS address identifying the memory location within the local storage of this SPU to which the data are to be read when the data become current, are entered into the additional memory segment.

An additional memory segment also is allocated for each memory location within the local storage of the SPUs. This additional memory segment stores one bit, designated the "busy bit." The busy bit is used to reserve the associated LS memory location for the storage of specific data to be retrieved from the DRAM. If the busy bit is set to **1** for a particular memory location in local storage, the SPU can use this memory location only for the writing of

these specific data. On the other hand, if the busy bit is set to 0 for a particular memory location in local storage, the SPU can use this memory location for the writing of any data.

5           Examples of the manner in which the F/E bit, the SPU ID, the LS address and the busy bit are used to synchronize the reading and writing of data from and to the shared DRAM of a PU are illustrated in **Figures. 17-31.**

10           As shown in **Figure 17**, one or more PUs, e.g., PE **1720**, interact with DRAM **1702**. PE **1720** includes SPU **1722** and SPU **1740**. SPU **1722** includes control logic **1724**, and SPU **1740** includes control logic **1742**. SPU **1722** also includes local storage **1726**. This local storage includes a plurality of addressable memory locations **1728**. SPU **1740** includes local storage **1744**, and this local storage also includes a plurality of addressable memory locations **1746**. All of these addressable memory locations preferably are **1024** bits in size.

20           An additional segment of memory is associated with each LS addressable memory location. For example, memory segments **1729** and **1734** are associated with, respectively, local memory locations **1731** and **1732**, and memory segment **1752** is associated with local memory location **1750**. A "busy bit," as discussed above, is stored in each of these additional memory segments. Local memory location **1732** is shown with several Xs to indicate that this location contains data.



DRAM **1702** contains a plurality of addressable memory locations **1704**, including memory locations **1706** and **1708**. These memory locations preferably also are **1024** bits in size. An additional segment of memory also is associated with each of these memory locations. For example, additional memory segment **1760** is associated with memory location **1706**, and additional memory segment **1762** is associated with memory location **1708**. Status information relating to the data stored in each memory location is stored in the memory segment associated with the memory location. This status information includes, as discussed above, the F/E bit, the SPU ID and the LS address. For example, for memory location **1708**, this status information includes F/E bit **1712**, SPU ID **1714** and LS address **1716**.

Using the status information and the busy bit, the synchronized reading and writing of data from and to the shared DRAM among the SPUs of a PU, or a group of PUs, can be achieved.

**Figure 18** illustrates the initiation of the synchronized writing of data from LS memory location **1732** of SPU **1722** to memory location **1708** of DRAM **1702**. Control **1724** of SPU **1722** initiates the synchronized writing of these data. Since memory location **1708** is empty, F/E bit **1712** is set to 0. As a result, the data in LS location **1732** can be written into memory location **1708**. If this bit were set to 1 to indicate that memory location **1708** is full and contains current, valid data, on the other hand, control **1722** would receive an error message and be prohibited from writing data into this memory location.

The result of the successful synchronized writing of the data into memory location **1708** is shown in **Figure 19**. The written data are stored in memory location **1708**, and F/E bit **1712** is set to 1. This setting indicates that  
5 memory location **1708** is full and that the data in this memory location are current and valid.

**Figure 20** illustrates the initiation of the synchronized reading of data from memory location **1708** of DRAM **1702** to LS memory location **1750** of local storage **1744**.  
10 To initiate this reading, the busy bit in memory segment **1752** of LS memory location **1750** is set to 1 to reserve this memory location for these data. The setting of this busy bit to 1 prevents SPU **1740** from storing other data in this memory location.

15 As shown in **Figure 21**, control logic **1742** next issues a synchronize read command for memory location **1708** of DRAM **1702**. Since F/E bit **1712** associated with this memory location is set to 1, the data stored in memory location **1708** are considered current and valid. As a result, in  
20 preparation for transferring the data from memory location **1708** to LS memory location **1750**, F/E bit **1712** is set to 0. This setting is shown in **Figure 22**. The setting of this bit to 0 indicates that, following the reading of these data, the data in memory location **1708** will be invalid.

25 As shown in **Figure 23**, the data within memory location **1708** next are read from memory location **1708** to LS memory location **1750**. **Figure 24** shows the final state. A copy of the data in memory location **1708** is stored in LS memory location **1750**. F/E bit **1712** is set to 0 to indicate that

the data in memory location **1708** are invalid. This invalidity is the result of alterations to these data to be made by SPU **1740**. The busy bit in memory segment **1752** also is set to 0. This setting indicates that LS memory location **1750** now is available to SPU **1740** for any purpose, i.e., this LS memory location no longer is in a reserved state waiting for the receipt of specific data. LS memory location **1750**, therefore, now can be accessed by SPU **1740** for any purpose.

**Figures. 25-31** illustrate the synchronized reading of data from a memory location of DRAM **1702**, e.g., memory location **1708**, to an LS memory location of an SPU's local storage, e.g., LS memory location **1752** of local storage **1744**, when the F/E bit for the memory location of DRAM **1702** is set to 0 to indicate that the data in this memory location are not current or valid. As shown in **Figure 25**, to initiate this transfer, the busy bit in memory segment **1752** of LS memory location **1750** is set to 1 to reserve this LS memory location for this transfer of data. As shown in **Figure 26**, control logic **1742** next issues a synchronize read command for memory location **1708** of DRAM **1702**. Since the F/E bit associated with this memory location, F/E bit **1712**, is set to 0, the data stored in memory location **1708** are invalid. As a result, a signal is transmitted to control logic **1742** to block the immediate reading of data from this memory location.

As shown in **Figure 27**, the SPU ID **1714** and LS address **1716** for this read command next are written into memory segment **1762**. In this case, the SPU ID for SPU **1740** and the LS memory location for LS memory location **1750** are written

into memory segment **1762**. When the data within memory location **1708** become current, therefore, this SPU ID and LS memory location are used for determining the location to which the current data are to be transmitted.

5           The data in memory location **1708** become valid and current when an SPU writes data into this memory location. The synchronized writing of data into memory location **1708** from, e.g., memory location **1732** of SPU **1722**, is illustrated in **Figure 28**. This synchronized writing of  
10 these data is permitted because F/E bit **1712** for this memory location is set to 0.

As shown in **Figure 29**, following this writing, the data in memory location **1708** become current and valid. SPU ID **1714** and LS address **1716** from memory segment **1762**,  
15 therefore, immediately are read from memory segment **1762**, and this information then is deleted from this segment. F/E bit **1712** also is set to 0 in anticipation of the immediate reading of the data in memory location **1708**. As shown in **Figure 30**, upon reading SPU ID **1714** and LS address **1716**,  
20 this information immediately is used for reading the valid data in memory location **1708** to LS memory location **1750** of SPU **1740**. The final state is shown in **Figure 31**. This figure shows the valid data from memory location **1708** copied to memory location **1750**, the busy bit in memory  
25 segment **1752** set to 0 and F/E bit **1712** in memory segment **1762** set to 0. The setting of this busy bit to 0 enables LS memory location **1750** now to be accessed by SPU **1740** for any purpose. The setting of this F/E bit to 0 indicates that the data in memory location **1708** no longer are current and  
30 valid.

**Figure 32** summarizes the operations described above and the various states of a memory location of the DRAM based upon the states of the F/E bit, the SPU ID and the LS address stored in the memory segment corresponding to the memory location. The memory location can have three states. These three states are an empty state **3280** in which the F/E bit is set to 0 and no information is provided for the SPU ID or the LS address, a full state **3282** in which the F/E bit is set to 1 and no information is provided for the SPU ID or LS address and a blocking state **3284** in which the F/E bit is set to 0 and information is provided for the SPU ID and LS address.

As shown in this figure, in empty state **3280**, a synchronized writing operation is permitted and results in a transition to full state **3282**. A synchronized reading operation, however, results in a transition to the blocking state **3284** because the data in the memory location, when the memory location is in the empty state, are not current.

In full state **3282**, a synchronized reading operation is permitted and results in a transition to empty state **3280**. On the other hand, a synchronized writing operation in full state **3282** is prohibited to prevent overwriting of valid data. If such a writing operation is attempted in this state, no state change occurs and an error message is transmitted to the SPU's corresponding control logic.

In blocking state **3284**, the synchronized writing of data into the memory location is permitted and results in a transition to empty state **3280**. On the other hand, a synchronized reading operation in blocking state **3284** is

prohibited to prevent a conflict with the earlier  
synchronized reading operation which resulted in this  
state. If a synchronized reading operation is attempted in  
blocking state **3284**, no state change occurs and an error  
5 message is transmitted to the SPU's corresponding control  
logic.

The scheme described above for the synchronized  
reading and writing of data from and to the shared DRAM  
also can be used for eliminating the computational  
10 resources normally dedicated by a processor for reading  
data from, and writing data to, external devices. This  
input/output (I/O) function could be performed by a PU.  
However, using a modification of this synchronization  
scheme, an SPU running an appropriate program can perform  
15 this function. For example, using this scheme, a PU  
receiving an interrupt request for the transmission of data  
from an I/O interface initiated by an external device can  
delegate the handling of this request to this SPU. The SPU  
then issues a synchronize write command to the I/O  
20 interface. This interface in turn signals the external  
device that data now can be written into the DRAM. The SPU  
next issues a synchronize read command to the DRAM to set  
the DRAM's relevant memory space into a blocking state. The  
SPU also sets to 1 the busy bits for the memory locations  
25 of the SPU's local storage needed to receive the data. In  
the blocking state, the additional memory segments  
associated with the DRAM's relevant memory space contain  
the SPU's ID and the address of the relevant memory  
locations of the SPU's local storage. The external device  
30 next issues a synchronize write command to write the data  
directly to the DRAM's relevant memory space. Since this

memory space is in the blocking state, the data are immediately read out of this space into the memory locations of the SPU's local storage identified in the additional memory segments. The busy bits for these memory locations then are set to 0. When the external device completes writing of the data, the SPU issues a signal to the PU that the transmission is complete.

Using this scheme, therefore, data transfers from external devices can be processed with minimal computational load on the PU. The SPU delegated this function, however, should be able to issue an interrupt request to the PU, and the external device should have direct access to the DRAM.

The DRAM of each PU includes a plurality of "sandboxes." A sandbox defines an area of the shared DRAM beyond which a particular SPU, or set of SPUs, cannot read or write data. These sandboxes provide security against the corruption of data being processed by one SPU by data being processed by another SPU. These sandboxes also permit the downloading of software cells from network 104 into a particular sandbox without the possibility of the software cell corrupting data throughout the DRAM. In the present invention, the sandboxes are implemented in the hardware of the DRAMs and DMACs. By implementing these sandboxes in this hardware rather than in software, advantages in speed and security are obtained.

The PU of a PU controls the sandboxes assigned to the SPUs. Since the PU normally operates only trusted programs, such as an operating system, this scheme does not

jeopardize security. In accordance with this scheme, the PU builds and maintains a key control table. This key control table is illustrated in **Figure 33**. As shown in this figure, each entry in key control table **3302** contains an identification (ID) **3304** for an SPU, an SPU key **3306** for that SPU and a key mask **3308**. The use of this key mask is explained below. Key control table **3302** preferably is stored in a relatively fast memory, such as a static random access memory (SRAM), and is associated with the DMAC. The entries in key control table **3302** are controlled by the PU. When an SPU requests the writing of data to, or the reading of data from, a particular storage location of the DRAM, the DMAC evaluates the SPU key **3306** assigned to that SPU in key control table **3302** against a memory access key associated with that storage location.

As shown in **Figure 34**, a dedicated memory segment **3410** is assigned to each addressable storage location **3406** of a DRAM **3402**. A memory access key **3412** for the storage location is stored in this dedicated memory segment. As discussed above, a further additional dedicated memory segment **3408**, also associated with each addressable storage location **3406**, stores synchronization information for writing data to, and reading data from, the storage-location.

In operation, an SPU issues a DMA command to the DMAC. This command includes the address of a storage location **3406** of DRAM **3402**. Before executing this command, the DMAC looks up the requesting SPU's key **3306** in key control table **3302** using the SPU's ID **3304**. The DMAC then compares the SPU key **3306** of the requesting SPU to the memory access key



3412 stored in the dedicated memory segment 3410 associated with the storage location of the DRAM to which the SPU seeks access. If the two keys do not match, the DMA command is not executed. On the other hand, if the two keys match,  
5 the DMA command proceeds and the requested memory access is executed.

An alternative embodiment is illustrated in **Figure 35**. In this embodiment, the PU also maintains a memory access control table 3502. Memory access control table 3502  
10 contains an entry for each sandbox within the DRAM. In the particular example of **Figure 35**, the DRAM contains 64 sandboxes. Each entry in memory access control table 3502 contains an identification (ID) 3504 for a sandbox, a base memory address 3506, a sandbox size 3508, a memory access  
15 key 3510 and an access key mask 3512. Base memory address 3506 provides the address in the DRAM which starts a particular memory sandbox. Sandbox size 3508 provides the size of the sandbox and, therefore, the endpoint of the particular sandbox.

20 **Figure 36** is a flow diagram of the steps for executing a DMA command using key control table 3302 and memory access control table 3502. In step 3602, an SPU issues a DMA command to the DMAC for access to a particular memory location or locations within a sandbox. This command  
25 includes a sandbox ID 3504 identifying the particular sandbox for which access is requested. In step 3604, the DMAC looks up the requesting SPU's key 3306 in key control table 3302 using the SPU's ID 3304. In step 3606, the DMAC uses the sandbox ID 3504 in the command to look up in  
30 memory access control table 3502 the memory access key 3510

associated with that sandbox. In step **3608**, the DMAC compares the SPU key **3306** assigned to the requesting SPU to the access key **3510** associated with the sandbox. In step **3610**, a determination is made of whether the two keys  
5 match. If the two keys do not match, the process moves to step **3612** where the DMA command does not proceed and an error message is sent to either the requesting SPU, the PU or both. On the other hand, if at step **3610** the two keys are found to match, the process proceeds to step **3614** where  
10 the DMAC executes the DMA command.

The key masks for the SPU keys and the memory access keys provide greater flexibility to this system. A key mask for a key converts a masked bit into a wildcard. For example, if the key mask **3308** associated with an SPU key  
15 **3306** has its last two bits set to "mask," designated by, e.g., setting these bits in key mask **3308** to 1, the SPU key can be either a 1 or a 0 and still match the memory access key. For example, the SPU key might be **1010**. This SPU key normally allows access only to a sandbox having an access  
20 key of **1010**. If the SPU key mask for this SPU key is set to **0001**, however, then this SPU key can be used to gain access to sandboxes having an access key of either **1010** or **1011**. Similarly, an access key **1010** with a mask set to **0001** can  
25 be accessed by an SPU with an SPU key of either **1010** or **1011**. Since both the SPU key mask and the memory key mask can be used simultaneously, numerous variations of accessibility by the SPUs to the sandboxes can be established.

The present invention also provides a new programming  
30 model for the processors of system **101**. This programming

model employs software cells **102**. These cells can be transmitted to any processor on network **104** for processing. This new programming model also utilizes the unique modular architecture of system **101** and the processors of system **101**.

Software cells are processed directly by the SPUs from the SPU's local storage. The SPUs do not directly operate on any data or programs in the DRAM. Data and programs in the DRAM are read into the SPU's local storage before the SPU processes these data and programs. The SPU's local storage, therefore, includes a program counter, stack and other software elements for executing these programs. The PU controls the SPUs by issuing direct memory access (DMA) commands to the DMAC.

The structure of software cells **102** is illustrated in **Figure 37**. As shown in this figure, a software cell, e.g., software cell **3702**, contains routing information section **3704** and body **3706**. The information contained in routing information section **3704** is dependent upon the protocol of network **104**. Routing information section **3704** contains header **3708**, destination ID **3710**, source ID **3712** and reply ID **3714**. The destination ID includes a network address. Under the TCP/IP protocol, e.g., the network address is an Internet protocol (IP) address. Destination ID **3710** further includes the identity of the PU and SPU to which the cell should be transmitted for processing. Source ID **3712** contains a network address and identifies the PU and SPU from which the cell originated to enable the destination PU and SPU to obtain additional information regarding the cell if necessary. Reply ID **3714** contains a network address and

identifies the PU and SPU to which queries regarding the cell, and the result of processing of the cell, should be directed.

5 Cell body **3706** contains information independent of the network's protocol. The exploded portion of **Figure 37** shows the details of cell body **3706**. Header **3720** of cell body **3706** identifies the start of the cell body. Cell interface **3722** contains information necessary for the cell's utilization. This information includes global unique ID  
10 **3724**, required SPUs **3726**, sandbox size **3728** and previous cell ID **3730**.

Global unique ID **3724** uniquely identifies software cell **3702** throughout network **104**. Global unique ID **3724** is generated on the basis of source ID **3712**, e.g. the unique  
15 identification of a PU or SPU within source ID **3712**, and the time and date of generation or transmission of software cell **3702**. Required SPUs **3726** provides the minimum number of SPUs required to execute the cell. Sandbox size **3728**  
20 provides the amount of protected memory in the required SPUs' associated DRAM necessary to execute the cell. Previous cell ID **3730** provides the identity of a previous cell in a group of cells requiring sequential execution, e.g., streaming data.

Implementation section **3732** contains the cell's core  
25 information. This information includes DMA command list **3734**, programs **3736** and data **3738**. Programs **3736** contain the programs to be run by the SPUs (called "spulets"), e.g., SPU programs **3760** and **3762**, and data **3738** contain the data to be processed with these programs. DMA command list

**3734** contains a series of DMA commands needed to start the programs. These DMA commands include DMA commands **3740**, **3750**, **3755** and **3758**. The PU issues these DMA commands to the DMAC.

5 DMA command **3740** includes VID **3742**. VID **3742** is the virtual ID of an SPU which is mapped to a physical ID when the DMA commands are issued. DMA command **3740** also includes load command **3744** and address **3746**. Load command **3744** directs the SPU to read particular information from the  
10 DRAM into local storage. Address **3746** provides the virtual address in the DRAM containing this information. The information can be, e.g., programs from programs section **3736**, data from data section **3738** or other data. Finally, DMA command **3740** includes local storage address **3748**. This  
15 address identifies the address in local storage where the information should be loaded. DMA commands **3750** contain similar information. Other DMA commands are also possible.

DMA command list **3734** also includes a series of kick commands, e.g., kick commands **3755** and **3758**. Kick commands  
20 are commands issued by a PU to an SPU to initiate the processing of a cell. DMA kick command **3755** includes virtual SPU ID **3752**, kick command **3754** and program counter **3756**. Virtual SPU ID **3752** identifies the SPU to be kicked, kick command **3754** provides the relevant kick command and  
25 program counter **3756** provides the address for the program counter for executing the program. DMA kick command **3758** provides similar information for the same SPU or another SPU.

As noted, the PUs treat the SPUs as independent processors, not co-processors. To control processing by the SPUs, therefore, the PU uses commands analogous to remote procedure calls. These commands are designated "SPU Remote Procedure Calls" (SRPCs). A PU implements an SRPC by issuing a series of DMA commands to the DMAC. The DMAC loads the SPU program and its associated stack frame into the local storage of an SPU. The PU then issues an initial kick to the SPU to execute the SPU Program.

**Figure 38** illustrates the steps of an SRPC for executing an spulet. The steps performed by the PU in initiating processing of the spulet by a designated SPU are shown in the first portion **3802** of **Figure 38**, and the steps performed by the designated SPU in processing the spulet are shown in the second portion **3804** of **Figure 38**.

In step **3810**, the PU evaluates the spulet and then designates an SPU for processing the spulet. In step **3812**, the PU allocates space in the DRAM for executing the spulet by issuing a DMA command to the DMAC to set memory access keys for the necessary sandbox or sandboxes. In step **3814**, the PU enables an interrupt request for the designated SPU to signal completion of the spulet. In step **3818**, the PU issues a DMA command to the DMAC to load the spulet from the DRAM to the local storage of the SPU. In step **3820**, the DMA command is executed, and the spulet is read from the DRAM to the SPU's local storage. In step **3822**, the PU issues a DMA command to the DMAC to load the stack frame associated with the spulet from the DRAM to the SPU's local storage. In step **3823**, the DMA command is executed, and the stack frame is read from the DRAM to the SPU's local

storage. In step **3824**, the PU issues a DMA command for the DMAC to assign a key to the SPU to allow the SPU to read and write data from and to the hardware sandbox or sandboxes designated in step **3812**. In step **3826**, the DMAC  
5 updates the key control table (KTAB) with the key assigned to the SPU. In step **3828**, the PU issues a DMA command "kick" to the SPU to start processing of the program. Other DMA commands may be issued by the PU in the execution of a particular SRPC depending upon the particular spulet.

10 As indicated above, second portion **3804** of **Figure 38** illustrates the steps performed by the SPU in executing the spulet. In step **3830**, the SPU begins to execute the spulet in response to the kick command issued at step **3828**. In step **3832**, the SPU, at the direction of the spulet,  
15 evaluates the spulet's associated stack frame. In step **3834**, the SPU issues multiple DMA commands to the DMAC to load data designated as needed by the stack frame from the DRAM to the SPU's local storage. In step **3836**, these DMA commands are executed, and the data are read from the DRAM  
20 to the SPU's local storage. In step **3838**, the SPU executes the spulet and generates a result. In step **3840**, the SPU issues a DMA command to the DMAC to store the result in the DRAM. In step **3842**, the DMA command is executed and the result of the spulet is written from the SPU's local  
25 storage to the DRAM. In step **3844**, the SPU issues an interrupt request to the PU to signal that the SRPC has been completed.

The ability of SPUs to perform tasks independently under the direction of a PU enables a PU to dedicate a  
30 group of SPUs, and the memory resources associated with a

group of SPUs, to performing extended tasks. For example, a PU can dedicate one or more SPUs, and a group of memory sandboxes associated with these one or more SPUs, to receiving data transmitted over network **104** over an extended period and to directing the data received during this period to one or more other SPUs and their associated memory sandboxes for further processing. This ability is particularly advantageous to processing streaming data transmitted over network **104**, e.g., streaming MPEG or streaming ATRAC audio or video data. A PU can dedicate one or more SPUs and their associated memory sandboxes to receiving these data and one or more other SPUs and their associated memory sandboxes to decompressing and further processing these data. In other words, the PU can establish a dedicated pipeline relationship among a group of SPUs and their associated memory sandboxes for processing such data.

In order for such processing to be performed efficiently, however, the pipeline's dedicated SPUs and memory sandboxes should remain dedicated to the pipeline during periods in which processing of spulets comprising the data stream does not occur. In other words, the dedicated SPUs and their associated sandboxes should be placed in a reserved state during these periods. The reservation of an SPU and its associated memory sandbox or sandboxes upon completion of processing of an spulet is called a "resident termination." A resident termination occurs in response to an instruction from a PU.

**Figures. 39, 40A and 40B** illustrate the establishment of a dedicated pipeline structure comprising a group of SPUs and their associated sandboxes for the processing of



streaming data, e.g., streaming MPEG data. As shown in **Figur 39**, the components of this pipeline structure include PE **3902** and DRAM **3918**. PE **3902** includes PU **3904**, DMAC **3906** and a plurality of SPUs, including SPU **3908**, SPU **3910** and SPU **3912**. Communications among PU **3904**, DMAC **3906** and these SPUs occur through PE bus **3914**. Wide bandwidth bus **3916** connects DMAC **3906** to DRAM **3918**. DRAM **3918** includes a plurality of sandboxes, e.g., sandbox **3920**, sandbox **3922**, sandbox **3924** and sandbox **3926**.

**Figure 40A** illustrates the steps for establishing the dedicated pipeline. In step **4010**, PU **3904** assigns SPU **3908** to process a network spulet. A network spulet comprises a program for processing the network protocol of network **104**. In this case, this protocol is the Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP data packets conforming to this protocol are transmitted over network **104**. Upon receipt, SPU **3908** processes these packets and assembles the data in the packets into software cells **102**. In step **4012**, PU **3904** instructs SPU **3908** to perform resident terminations upon the completion of the processing of the network spulet. In step **4014**, PU **3904** assigns PUs **3910** and **3912** to process MPEG spulets. In step **4015**, PU **3904** instructs SPUs **3910** and **3912** also to perform resident terminations upon the completion of the processing of the MPEG spulets. In step **4016**, PU **3904** designates sandbox **3920** as a source sandbox for access by SPU **3908** and SPU **3910**. In step **4018**, PU **3904** designates sandbox **3922** as a destination sandbox for access by SPU **3910**. In step **4020**, PU **3904** designates sandbox **3924** as a source sandbox for access by SPU **3908** and SPU **3912**. In step **4022**, PU **3904** designates sandbox **3926** as a destination sandbox for access by SPU

3912. In step 4024, SPU 3910 and SPU 3912 send synchronize read commands to blocks of memory within, respectively, source sandbox 3920 and source sandbox 3924 to set these blocks of memory into the blocking state. The process  
5 finally moves to step 4028 where establishment of the dedicated pipeline is complete and the resources dedicated to the pipeline are reserved. SPUs 3908, 3910 and 3912 and their associated sandboxes 3920, 3922, 3924 and 3926, therefore, enter the reserved state.

10       **Figure 40B** illustrates the steps for processing streaming MPEG data by this dedicated pipeline. In step 4030, SPU 3908, which processes the network spulet, receives in its local storage TCP/IP data packets from network 104. In step 4032, SPU 3908 processes these TCP/IP  
15 data packets and assembles the data within these packets into software cells 102. In step 4034, SPU 3908 examines header 3720 (**Figure 37**) of the software cells to determine whether the cells contain MPEG data. If a cell does not contain MPEG data, then, in step 4036, SPU 3908 transmits  
20 the cell to a general purpose sandbox designated within DRAM 3918 for processing other data by other SPUs not included within the dedicated pipeline. SPU 3908 also notifies PU 3904 of this transmission.

On the other hand, if a software cell contains MPEG  
25 data, then, in step 4038, SPU 3908 examines previous cell ID 3730 (**Figure 37**) of the cell to identify the MPEG data stream to which the cell belongs. In step 4040, SPU 3908 chooses an SPU of the dedicated pipeline for processing of the cell. In this case, SPU 3908 chooses SPU 3910 to  
30 process these data. This choice is based upon previous cell

ID **3730** and load balancing factors. For example, if previous cell ID **3730** indicates that the previous software cell of the MPEG data stream to which the software cell belongs was sent to SPU **3910** for processing, then the present software cell normally also will be sent to SPU **3910** for processing. In step **4042**, SPU **3908** issues a synchronize write command to write the MPEG data to sandbox **3920**. Since this sandbox previously was set to the blocking state, the MPEG data, in step **4044**, automatically is read from sandbox **3920** to the local storage of SPU **3910**. In step **4046**, SPU **3910** processes the MPEG data in its local storage to generate video data. In step **4048**, SPU **3910** writes the video data to sandbox **3922**. In step **4050**, SPU **3910** issues a synchronize read command to sandbox **3920** to prepare this sandbox to receive additional MPEG data. In step **4052**, SPU **3910** processes a resident termination. This processing causes this SPU to enter the reserved state during which the SPU waits to process additional MPEG data in the MPEG data stream.

Other dedicated structures can be established among a group of SPUs and their associated sandboxes for processing other types of data. For example, as shown in **Figure 41**, a dedicated group of SPUs, e.g., SPUs **4102**, **4108** and **4114**, can be established for performing geometric transformations upon three dimensional objects to generate two dimensional display lists. These two dimensional display lists can be further processed (rendered) by other SPUs to generate pixel data. To perform this processing, sandboxes are dedicated to SPUs **4102**, **4108** and **4114** for storing the three dimensional objects and the display lists resulting from the processing of these objects. For example, source

sandboxes **4104**, **4110** and **4116** are dedicated to storing the three dimensional objects processed by, respectively, SPU **4102**, SPU **4108** and SPU **4114**. In a similar manner, destination sandboxes **4106**, **4112** and **4118** are dedicated to  
5 storing the display lists resulting from the processing of these three dimensional objects by, respectively, SPU **4102**, SPU **4108** and SPU **4114**.

Coordinating SPU **4120** is dedicated to receiving in its local storage the display lists from destination sandboxes  
10 **4106**, **4112** and **4118**. SPU **4120** arbitrates among these display lists and sends them to other SPUs for the rendering of pixel data.

The processors of system **101** also employ an absolute timer. The absolute timer provides a clock signal to the  
15 SPUs and other elements of a PU which is both independent of, and faster than, the clock signal driving these elements. The use of this absolute timer is illustrated in **Figure 42**.

As shown in this figure, the absolute timer  
20 establishes a time budget for the performance of tasks by the SPUs. This time budget provides a time for completing these tasks which is longer than that necessary for the SPUs' processing of the tasks. As a result, for each task, there is, within the time budget, a busy period and a  
25 standby period. All spulets are written for processing on the basis of this time budget regardless of the SPUs' actual processing time or speed.

For example, for a particular SPU of a PU, a particular task may be performed during busy period **4202** of time budget **4204**. Since busy period **4202** is less than time budget **4204**, a standby period **4206** occurs during the time budget. During this standby period, the SPU goes into a sleep mode during which less power is consumed by the SPU.

The results of processing a task are not expected by other SPUs, or other elements of a PU, until a time budget **4204** expires. Using the time budget established by the absolute timer, therefore, the results of the SPUs' processing always are coordinated regardless of the SPUs' actual processing speeds.

In the future, the speed of processing by the SPUs will become faster. The time budget established by the absolute timer, however, will remain the same. For example, as shown in **Figure 42**, an SPU in the future will execute a task in a shorter period and, therefore, will have a longer standby period. Busy period **4208**, therefore, is shorter than busy period **4202**, and standby period **4210** is longer than standby period **4206**. However, since programs are written for processing on the basis of the same time budget established by the absolute timer, coordination of the results of processing among the SPUs is maintained. As a result, faster SPUs can process programs written for slower SPUs without causing conflicts in the times at which the results of this processing are expected.

In lieu of an absolute timer to establish coordination among the SPUs, the PU, or one or more designated SPUs, can analyze the particular instructions or microcode being

executed by an SPU in processing an spulet for problems in the coordination of the SPUs' parallel processing created by enhanced or different operating speeds. "No operation" ("NOOP") instructions can be inserted into the instructions and executed by some of the SPUs to maintain the proper sequential completion of processing by the SPUs expected by the spulet. By inserting these NOOPs into the instructions, the correct timing for the SPUs' execution of all instructions can be maintained.

**Figure 43** is a block diagram illustrating a processing element having a main processor and a plurality of secondary processors sharing a system memory. Processor Element (PE) **4305** includes processing unit (PU) **4310**, which, in one embodiment, acts as the main processor and runs an operating system. Processing unit **4310** may be, for example, a Power PC core executing a Linux operating system. PE **4305** also includes a plurality of synergistic processing complex's (SPCs) such as SPCs **4345**, **4365**, and **4385**. The SPCs include synergistic processing units (SPUs) that act as secondary processing units to PU **4310**, a memory storage unit, and local storage. For example, SPC **4345** includes SPU **4360**, MMU **4355**, and local storage **4359**; SPC **4365** includes SPU **4370**, MMU **4375**, and local storage **4379**; and SPC **4385** includes SPU **4390**, MMU **4395**, and local storage **4399**.

Each SPC may be configured to perform a different task, and accordingly, in one embodiment, each SPC may be accessed using different instruction sets. If PE **4305** is being used in a wireless communications system, for example, each SPC may be responsible for separate

processing tasks, such as modulation, chip rate processing, encoding, network interfacing, etc. In another embodiment, the SPCs may have identical instruction sets and may be used in parallel with each other to perform operations benefiting from parallel processing.

PE **4305** may also include level 2 cache, such as L2 cache **4315**, for the use of PU **4310**. In addition, PE **4305** includes system memory **4320**, which is shared between PU **4310** and the SPUs. System memory **4320** may store, for example, an image of the running operating system (which may include the kernel), device drivers, I/O configuration, etc., executing applications, as well as other data. System memory **4320** includes the local storage units of one or more of the SPCs, which are mapped to a region of system memory **4320**. For example, local storage **4359** may be mapped to mapped region **4335**, local storage **4379** may be mapped to mapped region **4340**, and local storage **4399** may be mapped to mapped region **4342**. PU **4310** and the SPCs communicate with each other and system memory **4320** through bus **4317** that is configured to pass data between these devices.

The MMUs are responsible for transferring data between an SPU's local store and the system memory. In one embodiment, an MMU includes a direct memory access (DMA) controller configured to perform this function. PU **4310** may program the MMUs to control which memory regions are available to each of the MMUs. By changing the mapping available to each of the MMUs, the PU may control which SPU has access to which region of system memory **4320**. In this manner, the PU may, for example, designate regions of the system memory as private for the exclusive use of a

particular SPU. In one embodiment, the SPUs' local stores may be accessed by PU **4310** as well as by the other SPUs using the memory map. In one embodiment, PU **4310** manages the memory map for the common system memory **4320** for all  
5 the SPUs. The memory map table may include PU **4310**'s L2 Cache **4315**, system memory **4320**, as well as the SPUs' shared local stores.

In one embodiment, the SPUs process data under the control of PU **4310**. The SPUs may be, for example, digital  
10 signal processing cores, microprocessor cores, micro controller cores, etc., or a combination of the above cores. Each one of the local stores is a storage area associated with a particular SPU. In one embodiment, each SPU can configure its local store as a private storage  
15 area, a shared storage area, or an SPU may configure its local store as a partly private and partly shared storage.

For example, if an SPU requires a substantial amount of local memory, the SPU may allocate 100% of its local store to private memory accessible only by that SPU. If,  
20 on the other hand, an SPU requires a minimal amount of local memory, the SPU may allocate 10% of its local store to private memory and the remaining 90% to shared memory. The shared memory is accessible by PU **4310** and by the other SPUs. An SPU may reserve part of its local store in order  
25 for the SPU to have fast, guaranteed memory access when performing tasks that require such fast access. The SPU may also reserve some of its local store as private when processing sensitive data, as is the case, for example, when the SPU is performing encryption/decryption.



**Figure 44** is a block diagram illustrating a processor element having a main processor and a plurality of secondary processors sharing a common system memory. Processor Element (PE) **4405** includes processing unit (PU) **4410**, which, in one embodiment, runs the operating system. Processing unit **4410** may be, for example, a Power PC core executing a Linux operating system. PE **4405** also includes a plurality of synergistic processing complex's (SPCs) such as SPCs **4445** through **4485**. Each SPC includes a synergistic processing unit (SPU), a memory storage unit, and local storage. For example, SPC **4445** includes SPU **4460**, MMU **4455**, and local storage **4459**; SPC **4465** includes SPU **4470**, MMU **4475**, and local storage **4479**; and SPC **4485** includes SPU **4490**, MMU **4495**, and local storage **4499**.

In one embodiment, the SPUs process data under the control of PU **4410**. The SPUs may be, for example, digital signal processing cores, microprocessor cores, micro controller cores, etc., or a combination of the above cores. In one embodiment, each one of the local stores is a storage area associated with a particular SPU. Each SPU can configure its local store as a private storage area, a shared storage area, or an SPU's local store may be partly private and partly shared.

For example, if an SPU requires a substantial amount of local memory, the SPU may allocate 100% of its local store to private memory accessible only by that SPU. If, on the other hand, an SPU requires a minimal amount of local memory, the SPU may allocate 10% of its local store to private memory and the remaining 90% to shared memory. The shared memory is accessible by PU **4410** and by the other

SPUs. An SPU may reserve part of its local store in order for the SPU to have fast, guaranteed access to some memory when performing tasks that require such fast access. The SPU may also reserve some of its local store as private  
5 when processing sensitive data, as is the case, for example, when the SPU is performing encryption/decryption.

The MMUs are responsible for transferring data between an SPU's local store and the system memory. In one embodiment, an MMU includes a direct memory access (DMA)  
10 controller configured to perform this function.

Each SPC may be set up to perform a different task, and accordingly, in one embodiment, each SPC may be accessed using different instruction sets. If PE **4405** is being used in a wireless communications system, for  
15 example, each SPC may be responsible for separate processing tasks, such as modulation, chip rate processing, encoding, network interfacing, etc. In another embodiment, each SPC may have identical instruction sets and may be used in parallel to perform operations benefiting from  
20 parallel processes.

The shared portion of the SPU's local stores may be accessed by PU **4410** as well as the other SPU's by mapping each shared region to system memory **4420**. In one embodiment, PU **4410** manages the memory map for the common  
25 system memory **4420**. The memory map table may include PU **4410**'s L2 Cache **4415**, system memory **4420**, as well as the SPU's shared memory.

A portion of system memory **4420** as shown is occupied by the operating system (OS **4425**), which includes kernel **4430** and loaded SPU device module **4435**. SPU device module **4435** is loaded in order to facilitate device-like access to the SPUs by executing applications, such as application **4440**. When an application such as application **4430** requests to send code and data to one of the SPUs, the kernel, using SPU device module **4435**, forwards the request to the appropriate using the appropriate SPU instructions. In a Unix environment, SPU **4450** may be accessed using the device access path, `"/dev/spu1"`, SPU **4470** may be accessed using the device access path, `"/dev/spu2"`, SPU **4490** may be accessed using the device access path, `"/dev/spuN"`, etc.

**Figure 45A** is a table illustrating a device configuration file that contains device access paths and corresponding devices. In one embodiment, Table **4510** contains a first column with a list of device access paths and a second column with a list of the actual device that corresponds each of the device paths. In a Unix environment, for example, the device access paths may be of the form, `"/dev/xxx"` where `"xxx"` is the name assigned to a device. `"/dev/spu1"`, for example, may correspond to SPU 1, `"/dev/spu2"` may correspond to SPU 2, etc. In addition, the configuration file may contain device access paths for other devices attached to the computer system such as disk drives, printers, etc. The device configuration file may be stored on disk, in conventional RAM, in nonvolatile RAM, ROM, etc.

**Figure 45B** is a table illustrating example application code for accessing a system's multiple processors. Table

**4520** shows example instructions by which an application can "write" data to a specific processor, "read" data from a specific processor, request a specific processor to process data, etc. In a Unix environment, an application may  
5 "write" data to the SPU 2, for example, by issuing the command: "write(/dev/spu2/, ...)". Similarly, a device may issue a read command (read(/dev/spu2/, ...)) as well other ioctl-type commands (ioctl(/dev/spu2/, ...)).

**Figure 46** is a flowchart illustrating a method for  
10 setting up device-like access to a system's multiple processors. Processing begins at **4600** whereupon, at step **4610**, the SPU device module is received and then loaded into operating system kernel **4620** in system memory **4620** of the computer system at step **4615**. The device module  
15 provides the kernel with the ability to communicate with the SPUs by providing the interface parameters between executing applications requesting access to SPUs and the SPUs.

At step **4625**, a device access path is created for  
20 accessing the first SPU (SPU 1). In a Unix environment, for example, the device access path, "/dev/spu1/" may be created for accessing SPU 1. Information on the device access path and the SPU corresponding to the device access path are saved in the device configuration file at step  
25 **4630**. A device configuration file is used for determining a correspondence between a device access path and the actual device for which the request is made.

A determination is then made as to whether more SPUs exist for which a device access path is required at

decision **4635**. If no more SPUs exist for which a device access path is required, decision **4635** branches to "no" branch **4645** whereupon processing ends at **4699**.

5 If more SPUs exist for which a device access path is required, decision **4635** branches to "yes" branch **4640** whereupon, at step **4650**, the next SPU requiring an access path is selected, and a device access path for that SPU is created. At step **4655**, information on the access path and the corresponding SPU are added to the device configuration  
10 file. Processing then loops back to decision **4635** where a determination is again made as to whether more SPUs exist requiring a device access path.

**Figure 47** is a flowchart illustrating a method by which an application requests device-like access to a  
15 processor in a multiprocessor system. Processing begins at step **4700** whereupon, at step **4710**, the computer system receives a request from a user to launch an application. In one embodiment, the computer system includes one or more processors, and the application is a multimedia application  
20 designed to take advantage of the computer system's multiple processors.

At step **4715**, the operating system kernel loads the application into system memory **4720** and initiates execution of the application. In one embodiment, the application  
25 executes on the main processor of the computer system. In another embodiment, the application may execute on one of the computer system's other processors.

At step **4725**, the kernel monitors whether an application currently executing is requesting, via a device call, for code to execute and data to be processed on a particular SPU. Applications can request access to a particular SPU similarly to the way applications would request access to other devices attached to the computer system such as disk drives, printers, etc. A processor device module (similar to a printer's or a disk drive's device module or driver) facilitates the execution of the application's request.

A determination is then made as to whether an application has issued such a request for one of the SPUs at decision **4730**. If an application has not issued a request for an SPU, decision **4730** branches to "no" branch **4740** whereupon processing returns to step **4725** where the kernel process continues to monitor for applications issuing a request for an SPU.

If an application has issued a request for an SPU, decision **4730** branches to "yes" branch **4740** whereupon, at step **4745**, the application's SPU instructions and data are placed for the SPU in system memory.

At step **4750**, the kernel, using the loaded SPU device module, determines which SPU is to receive the application's instructions and data and then communicates the instructions to the SPU. More details on the processing that takes place at step **4750** are provided in the flowchart of **Figure 48**.

**Figure 48** is a flowchart illustrating a method by which an operating system kernel receives a request from an application to access a processor in a multi-processor system and facilitates access to the processor. Processing begins at **4800** whereupon, at step **4820**, the kernel loads the SPU device module. The SPU device module facilitates a communication between the SPUs and an application requesting access to one of the SPUs. At step **4825**, the kernel using the device module reads the application's instructions/code from a region of system memory **4810** and identifies the target SPU.

At step **4830**, the kernel using the SPU device module and the instructions/code determines an instructions block for the identified SPU. The instructions block may include, for example, a pointer to the application's code, a pointer to the application's input data, and a pointer to an output buffer—a memory regions where the SPU is to place the processed data.

At step **4835**, a pointer to the instructions block is placed in the identified SPU's mailbox, which, in one embodiment, is located by the SPU. Mailboxes (such as mailbox **4815**) are memory spaces associated with each SPU used to facilitate handshaking between the SPUs and the PU/operating system. In one embodiment, each SPU periodically monitors its mailbox for pointers placed there by instructions placed there by the operating system kernel.

At step **4840**, the SPU reads the instructions and data from system memory **4810** and processes the data. More

5 details on the processing that takes place at step **4840** are provided in the flowchart of **Figure 49**. At step **4845**, the kernel is notified that the application's data have been processed by the SPU, and at step **4850**, the application reads the processed data from system memory **4810**. Processing ends at **4899**.

10 **Figure 49** is a flowchart illustrating a method with which a processor processes an application's request to access a processor. Processing begins at **4900** whereupon, at step **4910**, the SPU periodically monitors the SPU's mailbox to determine whether the operating system has requested the SPU's services. A determination is then made as to whether instructions for the SPU (or a pointer to such instructions) have been placed in the SPU's mailbox at decision **4915**. If instructions have not yet been placed in the SPU's mailbox, decision **4915** branches to "no" branch **4925** whereupon processing returns to step **4910** where the SPU continues to periodically monitor the SPU's mailbox.

20 If instructions have been placed in the SPU's mailbox, decision **4915** branches to "yes" branch **4925** whereupon, at step **4940**, the SPU accesses the SPU's mailbox and reads the instructions (or the pointer to the instructions placed in system memory). At step **4945**, the SPU, using DMA, transfers the instructions (prepared using the SPU device module) to the SPU's local store, and at step **4950**, the SPU, again using DMA, transfers the application data to be processed from system memory **4930** to the SPU's local store as well.



At step **4960**, the SPU processes the data placed there by the application. In one embodiment, as the SPU processes the data, the SPU places the data back in the SPU's local store. At step **4960**, the SPU transfers the  
5 processed data from the SPU's local store into the output buffer reserved for the SPU's processed data in system memory **4930**. The SPU may transfer the data periodically as the SPU processes the data or the SPU may transfer the data after the SPU has finished processing all the data.

10 At step **4965**, the SPU signals to the operating system the end of the data processing. The SPU may signal the kernel by placing an appropriate indicator in mailbox **4935** in an embodiment where the operating system is configured to periodically monitor the SPU mailboxes to determine  
15 whether one or more of the SPUs is attempting to communicate with the operating system. In another embodiment, the SPU may issue an appropriate interrupt, etc. Processing ends at **4999**.

While particular embodiments of the present invention  
20 have been shown and described, it will be obvious to those skilled in the art that, based upon the teachings herein, changes and modifications may be made without departing from this invention and its broader aspects and, therefore, the appended claims are to encompass within their scope all  
25 such changes and modifications as are within the true spirit and scope of this invention. Furthermore, it is to be understood that the invention is solely defined by the appended claims. It will be understood by those with skill in the art that if a specific number of an introduced claim  
30 element is intended, such intent will be explicitly recited

in the claim, and in the absence of such recitation no such limitation is present. For a non-limiting example, as an aid to understanding, the following appended claims contain usage of the introductory phrases "at least one" and "one  
5 or more" to introduce claim elements. However, the use of such phrases should not be construed to imply that the introduction of a claim element by the indefinite articles "a" or "an" limits any particular claim containing such introduced claim element to inventions containing only one  
10 such element, even when the same claim includes the introductory phrases "one or more" or "at least one" and indefinite articles such as "a" or "an"; the same holds true for the use in the claims of definite articles.